Projected Model Counting: Beyond Independent Support *

Jiong Yang[1], Supratik Chakraborty[2], and Kuldeep S. Meel[1]

[1] School of Computing, National University of Singapore, Singapore
[2] Indian Institute of Technology Bombay, India

Abstract. Given a system of constraints over a set X of variables, projected model counting asks us to count satisfying assignments of the constraint system projected on a subset $\mathcal{P}$ of X. A key idea used in modern projected counters is to first compute an *independent support*, say $\mathcal{I}$, that is often a small subset of $\mathcal{P}$, and to then count models projected on $\mathcal{I}$ instead of on $\mathcal{P}$. While this has been effective in scaling performance of counters, the question of whether we can benefit by projecting on variables beyond $\mathcal{P}$ has not been explored. In this paper, we study this question and show that contrary to intuition, it can be beneficial to project on variables even beyond $\mathcal{P}$. In several applications, a good upper bound of the projected model count often suffices. We show that in several such cases, we can identify a set of variables, called *upper bound support (UBS)*, that is not necessarily a subset of $\mathcal{P}$, and yet counting models projected on UBS guarantees an upper bound of the projected model count. Theoretically, a UBS can be exponentially smaller than the smallest independent support. Our experiments show that even otherwise, UBS-based projected counting can be faster than independent support-based projected counting, while yielding bounds of high quality. Based on extensive experiments, we find that UBS-based projected counting can solve many problem instances that are beyond the reach of a state-of-the-art independent support-based projected model counter.

1 Introduction

Given a Boolean formula φ over a set X of variables, and a subset $\mathcal{P}$ of X, the problem of projected model counting asks us to determine the number of satisfying assignments of φ projected on $\mathcal{P}$. Projected model counting is # NP-complete in general [33][3], and has several important applications ranging from verification of neural networks [4], hardware and software verification [32], reliability of power grids [11], probabilistic inference [25], and the like. This problem has therefore attracted significant attention from both theoreticians and practitioners over the years [7, 9, 18, 27, 28, 30, 34]. While an ideal projected model counter offers high scalability *and* strong quality guarantees for computed counts, these goals are

* The resulting tool is available open-source at `https://github.com/meelgroup/arjun`

[3] A special case where $\mathcal{P} = X$ is known to be #P-complete [34]

often hard to achieve simultaneously in practice. A pragmatic approach in several applications is therefore to use counters that offer good scalability and good quality of counts in practice, even if worst-case quality guarantees are weaker than ideal. Unfortunately, designing such counters is not easy either, and this motivates our current work.

Over the past decade, hashing-based techniques have emerged as a promising approach to projected model counting, since they scale moderately in practice, while providing strong approximation guarantees [6, 7, 13, 18, 27]. For propositional model counting, the hash functions are implemented using random XOR clauses over variables in $\mathcal{P}$. Starting from a formula φ in conjunctive normal form (CNF), these techniques construct a CNF+XOR formula φ' consisting of a conjunction of CNF clauses from φ and random XOR clauses implementing the hash functions. If each variable in $\mathcal{P}$ is chosen with probability $1/2$ the expected size of a random XOR clause is $|\mathcal{P}|/2$. If the projection set is large, this can indeed result in large XOR clauses – a known source of poor performance of modern SAT solvers on CNF+XOR formulas [8, 16]. Researchers have therefore explored the use of hash functions with *sparse* XOR clauses [1, 12, 16, 19, 23] with moderate success.

A practically effective idea to address the problem of large XOR clauses was introduced in [8], wherein the notion of an *independent support* $\mathcal{I}$ ($\subseteq \mathcal{P}$), was introduced. Specifically, it was shown in [8] that (a) random XOR clauses over $\mathcal{I}$ suffice to provide strong guarantees for computed bounds, and (b) for a large class of practical benchmarks, $|\mathcal{I}|$ is much smaller than $|\mathcal{P}|$. Hence, constructing random XOR clauses over $\mathcal{I}$ instead of over $\mathcal{P}$ reduces the expected size of a random XOR clause, thereby improving the runtime performance of hashing-based counters [19]. Subsequently, independent supports have also been found to be useful in the context of exact projected model counting [21, 22, 26].

The runtime performance improvements achieved by (projected) model counters over the past decade have significantly broadened the scope of their applications, which, in turn, has brought the focus sharply back on performance scalability. Importantly, for several crucial applications such as neural network verification [4], quantified information flow [5], software reliability [32], reliability of power grids [11], etc. we are primarily interested in good upper bound estimates of projected model counts. As aptly captured by Achlioptas and Theodoropoulos [1], while obtaining "lower bounds are easy" in the context of projected model counting, such is not the case for good upper bounds. Therefore, scaling up to large problem instances while obtaining good upper bound estimates remains an important challenge in this area.

The primary contribution of this paper is a new approach to selecting variables on which to project solutions, with the goal of improving scalability of hashing-based projected counters when good upper bounds of projected counts are of interest. Towards this end, we generalize the notion of an independent support $\mathcal{I}$. Specifically, we note that the restriction $\mathcal{I} \subseteq \mathcal{P}$ ensures a two-way implication: if two solutions agree on $\mathcal{I}$, then they also agree on $\mathcal{P}$, and vice-versa. Since we are interested in upper bounds, we relax this requirement to a one-sided

implication, i.e., we wish to find a set $\mathcal{U} \subseteq X$ (not necessarily a subset of $\mathcal{P}$) such that if two solutions agree on $\mathcal{U}$, then they agree on $\mathcal{P}$, but not necessarily vice versa. We call such a set $\mathcal{U}$ an *Upper Bound Support*, or UBS for short. We show that using random XOR clauses over UBS in hashing-based projected counting yields provable upper bounds of the projected counts. We also show some important properties of UBS, including an exponential gap between the smallest UBS and the smallest independent support for a class of problems. Our study suggests a simple algorithm, called FindUBS, to determine UBS, that can be fine-tuned heuristically.

To evaluate the effectiveness of our idea, we augment a state-of-the-art model counter, ApproxMC4, with UBS to obtain UBS+ApproxMC4. Through an extensive empirical evaluation on 2632 benchmark instances arising from diverse domains, we compare the performance of UBS+ApproxMC4 with IS+ApproxMC4, i.e. ApproxMC4 augmented with independent support computation. Our experiments show that UBS+ApproxMC4 is able to solve 208 more instances than IS+ApproxMC4. Furthermore, the geometric mean of the absolute value of log-ratio of counts returned by UBS+ApproxMC4 and IS+ApproxMC4 is 1.32, thereby validating the claim that using UBS can lead to empirically good upper bounds. In this context, it is worth remarking that a recent study [2] comparing different partition function[4] estimation techniques labeled a method with the absolute value of log-ratio of counts less than 5 as a *reliable method*.

The rest of the paper is organized as follows. We present notation and preliminaries in Section 2. To situate our contribution, we present a survey of related work in Section 3. We then present the primary technical contributions of our work, including the notion of UBS and an algorithmic procedure to determine UBS, in Section 4. We present our empirical evaluation in Section 5, and finally conclude in Section 6.

2 Notation and Preliminaries

Let $X = \{x_1, x_2 \dots x_n\}$ be a set of propositional variables appearing in a propositional formula φ. The set X is called the *support* of φ, and denoted $\mathsf{Sup}(\varphi)$. A *literal* is either a propositional variable or its negation. The formula φ is said to be in Conjunctive Normal Form (CNF) if φ is a conjunction of *clauses*, where each *clause* is disjunction of literals. An *assignment* σ of X is a mapping $X \to \{0,1\}$. If φ evaluates to 1 under assignment σ, we say that σ is a *model* or *satisfying assignment* of φ, and denote this by $\sigma \models \varphi$. For every $\mathcal{P} \subseteq X$, the *projection* of σ on $\mathcal{P}$, denoted $\sigma_{\downarrow \mathcal{P}}$, is a mapping $\mathcal{P} \to \{0,1\}$ such that $\sigma_{\downarrow \mathcal{P}}(v) = \sigma(v)$ for all $v \in \mathcal{P}$. Conversely we say that an assignment $\widehat{\sigma} : \mathcal{P} \to \{0,1\}$ can be *extended* to a model of φ if there exists a model σ of φ such that $\widehat{\sigma} = \sigma_{\downarrow \mathcal{P}}$. The set of all

[4] The problem of partition function estimation is known to be #P-complete and reduces to model counting; the state of the art techniques for partition function estimates are based on model counting [10].

models of φ is denoted $sol(\varphi)$, and the projection of this set on $\mathcal{P} \subseteq X$ is denoted $sol(\varphi)_{\downarrow\mathcal{P}}$. We call the set $\mathcal{P}$ a *projection set* in our subsequent discussion[5].

The problem of *projected model counting* is to compute $|sol(\varphi)_{\downarrow\mathcal{P}}|$ for a given CNF formula φ and projection set $\mathcal{P}$. An exact projected model counter is a deterministic algorithm that takes φ and $\mathcal{P}$ as inputs and returns $|sol(\varphi)_{\downarrow\mathcal{P}}|$ as output. A *probably approximately correct* (or PAC) projected model counter is a probabilistic algorithm that takes as additional inputs a tolerance $\varepsilon > 0$, and a confidence parameter $\delta \in (0, 1]$, and returns a count c such that $\mathsf{Pr}\Big[\frac{|sol(\varphi)_{\downarrow\mathcal{P}}|}{(1+\varepsilon)} \leq c \leq (1+\varepsilon) \cdot |sol(\varphi)_{\downarrow\mathcal{P}}|\Big] \geq 1 - \delta$, where $\mathsf{Pr}[E]$ denotes the probability of event E.

Definition 1. *Given a formula φ and a projection set $\mathcal{P} \subseteq \mathsf{Sup}(\varphi)$, a subset of variables $\mathcal{I} \subseteq \mathcal{P}$ is called an* independent support (IS) *of $\mathcal{P}$ in φ if for every $\sigma_1, \sigma_2 \in sol(\varphi)$, we have $\big(\sigma_{1\downarrow\mathcal{I}} = \sigma_{2\downarrow\mathcal{I}}\big) \Rightarrow \big(\sigma_{1\downarrow\mathcal{P}} = \sigma_{2\downarrow\mathcal{P}}\big)$.*

Since $\big(\sigma_{1\downarrow\mathcal{P}} = \sigma_{2\downarrow\mathcal{P}}\big) \Rightarrow \big(\sigma_{1\downarrow\mathcal{I}} = \sigma_{2\downarrow\mathcal{I}}\big)$ holds trivially when $\mathcal{I} \subseteq \mathcal{P}$, it follows from Definition 1 that if $\mathcal{I}$ is an independent support of $\mathcal{P}$ in φ, then $\big(\sigma_{1\downarrow\mathcal{I}} = \sigma_{2\downarrow\mathcal{I}}\big) \Leftrightarrow \big(\sigma_{1\downarrow\mathcal{P}} = \sigma_{2\downarrow\mathcal{P}}\big)$. Empirical studies have shown that the size of an independent support $\mathcal{I}$ is often significantly smaller than that of the original projection set $\mathcal{P}$ [8, 19, 21, 26]. In fact, the overhead of finding a small independent support $\mathcal{I}$ is often more than compensated by the efficiency obtained by counting projections of satisfying assignments on $\mathcal{I}$, instead of on the original projection set $\mathcal{P}$.

3 Related Work

As mentioned in Section 1, state-of-the-art hashing-based projected model counters work by adding random XOR clauses over the projection set $\mathcal{P}$ to a given CNF formula φ before finding satisfying assignments of the CNF+XOR formula. There are several inter-related factors that affect the runtime performance of such counters, and isolating the effect of any one factor is difficult. Nevertheless, finding satisfying assignments of the CNF+XOR formula is among the most significant bottlenecks. Among other things, the average size (i.e. number of literals) in XOR clauses correlates positively with the time taken to solve CNF+XOR formulas using modern conflict-driven clause learning (CDCL) SAT solvers [19].

The idea of using random XOR clauses over an independent support $\mathcal{I}$ ($\subseteq \mathcal{P}$) that is potentially much smaller than $\mathcal{P}$ was introduced in [8]. This is particularly effective when a small subset of variables functionally determines the large majority of variables in a formula, as happens, for example, when Tseitin encoding is used to transform a non-CNF formula to an equisatisfiable CNF formula. State-of-the-art hashing-based model counters, viz. ApproxMC4 [27], therefore routinely use random XOR clauses over the independent support. While the naive way of choosing each variable in $\mathcal{I}$ with probability $1/2$ gives a random XOR clause with expected size $|\mathcal{I}|/2$, specialized hash functions can also be

[5] Projection set has also been referred to as sampling set in prior work [8, 27].

defined such that the expected size of a random XOR clause is $p \cdot |\mathcal{I}|$, with $p < 1/2$ [1, 12, 23]. The works of [1, 12] achieved this goal while guaranteeing a constant factor approximation of the reported count. The work of [23] achieved a similar reduction in the expected size of XOR clauses, while guaranteeing PAC-style bounds.

All earlier work focused on random XOR clauses chosen over subsets of the projection set $\mathcal{P}$. While this is a natural choice, we break free from this restriction and allow XOR clauses to be constructed over any subset of variables as long as the model count projected on the chosen subset bounds the model count projected on $\mathcal{P}$ from above. This allows us more flexibility in constructing CNF+XOR formulas, which as our experiments confirm, leads to improved overall performance of projected model counting in several cases. Since we guarantee upper bounds of the desired counts, our approach yields an *upper bounding projected model counter*. Nevertheless, as our experiments show, the bounds obtained using our approach are consistently very close to the projected counts reported using independent support. Therefore, in practice, our approach gives high quality bounds on projected model counts more efficiently than state-of-the-art hashing-based techniques that use independent supports.

It is worth mentioning that several *bounding model counters* have been reported earlier in the literature. These counters produce a count that is at least as large (or, as small, as the case may be) as the true model count of a given CNF formula with a specified confidence. Notable examples are SampleCount [17], BPCount [20], MBound and Hybrid-MBound [18] and MiniCount [20]. Owing to several technical reasons, however, these bounding counters scale poorly compared to state-of-the-art hashing-based counters like ApproxMC4 [27] in practice. Unlike earlier bounding counters, we first carefully identify a subset of variables (not restricted to be a subset of $\mathcal{P}$), and then use state-of-the-art hashing-based approximate projected counting using this subset as the new projection set. Therefore, our approach directly benefits from improvements in performance of hashing-based projected counting achieved over the years. Furthermore, by carefully controlling the chosen subset of variables, we can also control the quality of the bound. As an extreme case, if all variables are chosen from $\mathcal{P}$, then our approach produces counts with true PAC-style guarantees.

4 Technical Contribution

In this section, we generalize the notion of independent support, and give technical details of projected model counting using this generalization.

Definition 2. *Given a CNF formula φ and a projection set $\mathcal{P}$, let $\mathcal{S} \subseteq \mathsf{Sup}(\varphi)$ be such that for every $\sigma_1, \sigma_2 \in sol(\varphi)$, we have $\left(\sigma_{1\downarrow\mathcal{S}} = \sigma_{2\downarrow\mathcal{S}}\right) \bowtie \left(\sigma_{1\downarrow\mathcal{P}} = \sigma_{2\downarrow\mathcal{P}}\right)$, where $\bowtie \ \in \{\Rightarrow, \Leftarrow, \Leftrightarrow\}$. Then $\mathcal{S}$ is called a*

1. generalized independent support (GIS) *of $\mathcal{P}$ in φ if $\bowtie$ is $\Leftrightarrow$*
2. upper bound support (UBS) *of $\mathcal{P}$ in φ if $\bowtie$ is $\Rightarrow$*
3. lower bound support (LBS) *of $\mathcal{P}$ in φ if $\bowtie$ is $\Leftarrow$*

Note that in the above definition, $\mathcal{S}$ need not be a subset of $\mathcal{P}$. In fact, if $\mathcal{S}$ is restricted to be a subset of $\mathcal{P}$, the definitions of GIS and UBS coincide with that of IS (Definition 1), while LBS becomes a trivial concept (every subset of $\mathcal{P}$ is indeed an LBS of $\mathcal{P}$ in φ). The following lemma now follows immediately.

Lemma 1. *Let $\mathcal{G}, \mathcal{U}$ and $\mathcal{L}$ be GIS, UBS and LBS, respectively, of $\mathcal{P}$ in φ. Then* $|sol(\varphi)_{\downarrow \mathcal{L}}| \leq |sol(\varphi)_{\downarrow \mathcal{P}}| = |sol(\varphi)_{\downarrow \mathcal{G}}| \leq |sol(\varphi)_{\downarrow \mathcal{U}}|$.

Let $\mathcal{UBS}, \mathcal{LBS}, \mathcal{GIS}$ and $\mathcal{IS}$ be the set of all UBS, LBS, GIS and IS respectively of a projection set $\mathcal{P}$ in φ. It is easy to see that $\mathcal{IS} \subseteq \mathcal{GIS} \subseteq \mathcal{UBS}$, and $\mathcal{GIS} \subseteq \mathcal{LBS}$. While each of the notions of GIS, UBS and LBS are of independent interest, this paper focuses primarily on UBS because we found this notion particularly useful in practical projected model counting. Additionally, as the above inclusion relations show, $\mathcal{UBS}$ and $\mathcal{LBS}$ are the largest classes among $\mathcal{UBS}, \mathcal{LBS}, \mathcal{GIS}$ and $\mathcal{IS}$; hence, finding an UBS is likely to be easier than finding a GIS. Furthermore, the notion of UBS continues to remain interesting (but not so for LBS) even when $\mathcal{I}$ is chosen to be a subset of $\mathcal{P}$.

We call a UBS $\mathcal{U}$ (resp. LBS $\mathcal{L}$, GIS $\mathcal{G}$ and IS $\mathcal{I}$) of $\mathcal{P}$ in φ *minimal* if there is no other UBS (resp. LBS, GIS and IS) of $\mathcal{P}$ in φ that is a strict subset of $\mathcal{U}$ (resp. of $\mathcal{L}$, $\mathcal{G}$ and $\mathcal{I}$).

Example 1. Consider a CNF formula $\varphi(x_1, x_2, x_3, x_4) \equiv (x_3 \vee x_4) \wedge (x_1 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_4 \vee \neg x_2)$. There are four satisfying assignments of φ, given by $(x_1, x_2, x_3, x_4) \in \{(0,0,1,1), (0,1,0,1), (1,0,1,1), (1,1,1,0)\}$. If $\mathcal{P} = \{x_1, x_3, x_4\}$, it can be seen that the only minimal IS of $\mathcal{P}$ in φ is $\{x_1, x_3, x_4\}$, whereas $\{x_1, x_2\}$ is a minimal UBS and also GIS of $\mathcal{P}$ in φ. Any single variable subset of $\{x_1, x_2, x_3, x_4\}$ serves as a minimal LBS of $\mathcal{P}$ in φ.

In the remainder of this section, we first explore some interesting theoretical properties of GIS and UBS, and then proceed to develop a practical algorithm for computing a UBS from a given formula φ and projection set $\mathcal{P}$. Finally, we present an algorithm for computing bounds of projected model counts using the UBS thus computed.

4.1 Extremal properties of GIS and UBS

We first show that by allowing variables on which to project to lie beyond the projection set $\mathcal{P}$, we can obtain an exponential reduction in the count of variables on which to project.

Theorem 1. *For every $n > 1$, there exists a propositional formula φ_n on $(n-1) + \lceil \log_2 n \rceil$ variables and a projection set $\mathcal{P}_n$ with $|\mathcal{P}_n| = n-1$ such that*
- *The smallest GIS of $\mathcal{P}_n$ in φ_n is of size $\lceil \log_2 n \rceil$.*
- *The smallest UBS of $\mathcal{P}_n$ in φ_n is of size $\lceil \log_2 n \rceil$.*
- *The smallest IS of $\mathcal{P}_n$ in φ_n is $\mathcal{P}_n$ itself, and hence of size $n-1$.*

Proof:

	x_1	x_2	$\cdots$	x_{n-1}	y_1	y_2	$\cdots$	$y_{\log_2 n}$
σ_0	0	0	$\cdots$	0	0	$\cdots$	0	0
σ_1	1	0	$\cdots$	0	0	$\cdots$	0	1
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
σ_{n-1}	0	0	$\cdots$	1	1	$\cdots$	1	1

For notational convenience, we assume n to be a power of 2. Consider a formula φ_n on propositional variables $x_1, \ldots x_{n-1}$, $y_1, \ldots y_{\log_2 n}$ with n satisfying assignments, say $\sigma_0, \ldots \sigma_{n-1}$, as shown in the table below. Thus, for all $i \in \{1, \ldots n-1\}$, the values of $y_1 \ldots y_{\log_2 n}$ in σ_i encode i in binary (with y_1 being the most significant bit), the value of x_i is 1, and the values of all other x_j's are 0. For the special satisfying assignment σ_0, the values of all variables are 0.

Let $\mathcal{P}_n = \{x_1, \ldots x_{n-1}\}$. Clearly, $|sol(\varphi_n)| = |sol(\varphi_n)_{\downarrow \mathcal{P}_n}| = n$. Now consider the set of variables $\mathcal{G}_n = \{y_1, \ldots y_{\log_2 n}\}$. It is easy to verify that for every pair of satisfying assignments σ_i, σ_j of φ_n, $\left(\sigma_{i\downarrow \mathcal{G}_n} = \sigma_{j\downarrow \mathcal{G}_n}\right) \Leftrightarrow \left(\sigma_{i\downarrow \mathcal{P}_n} = \sigma_{j\downarrow \mathcal{P}_n}\right)$. Therefore, $\mathcal{G}_n$ is a GIS, and hence also a UBS, of $\mathcal{P}_n$ in φ_n, and $|\mathcal{G}_n| = \log_2 n$. Indeed, specifying $y_1, \ldots y_{\log_2 n}$ completely specifies the value of all variables for every satisfying assignment of φ_n. Furthermore, since $|sol(\varphi_n)_{\downarrow \mathcal{P}_n}| = n$, every GIS and also UBS of $\mathcal{P}_n$ must be of size at least $\log_2 n$. Hence, $\mathcal{G}_n$ is a smallest-sized GIS, and also a smallest-sized UBS, of $\mathcal{P}_n$ in φ_n.

Let us now find how small an independent support (IS) of $\mathcal{P}_n$ in φ can be. Recall that $|sol(\varphi_n)_{\downarrow \mathcal{P}_n}| = n$. If possible, let there be an IS of $\mathcal{P}_n$, say $\mathcal{I}_n \subseteq \mathcal{P}_n$, where $|\mathcal{I}_n| < n-1$. Therefore, at least one variable in $\mathcal{P}_n$, say x_i, must be absent in $\mathcal{I}_n$. Now consider the satisfying assignments σ_i and σ_0. Clearly, both $\sigma_{i\downarrow \mathcal{I}_n}$ and $\sigma_{0\downarrow \mathcal{I}_n}$ are the all-0 vector of size $|\mathcal{I}_n|$. Therefore, $\sigma_{i\downarrow \mathcal{I}_n} = \sigma_{0\downarrow \mathcal{I}_n}$ although $\sigma_{i\downarrow \mathcal{P}_n} \neq \sigma_{0\downarrow \mathcal{P}_n}$. It follows that $\mathcal{I}_n$ cannot be an IS of $\mathcal{P}_n$ in φ_n. This implies that the smallest IS of $\mathcal{P}_n$ in φ_n is $\mathcal{P}_n$ itself, and has size $n-1$. □

Observe that the smallest GIS/UBS $\mathcal{G}_n$ above is disjoint from $\mathcal{P}_n$. Therefore, it can be beneficial to look outside the projection set when searching for a GIS or UBS. The next theorem shows that the opposite can also be true. The proof is deferred to the detailed technical report [**?**].

Theorem 2. *For every $n > 1$, there exist formulas φ_n and ψ_n on $(n-1) + \lceil \log_2 n \rceil$ variables and a projection set $\mathcal{Q}_n$ with $|\mathcal{Q}_n| = n - \lceil \log_2 n \rceil - 1$ such that the only GIS of $\mathcal{Q}_n$ in φ_n is $\mathcal{Q}_n$, and the smallest UBS of $\mathcal{Q}_n$ in ψ_n is also $\mathcal{Q}_n$.*

Theorems 1 and 2 indicate that the search for the smallest GIS or UBS is likely to be hard, since it has to potentially consider subsets of X ranging from those completely overlapping with $\mathcal{P}$ to those disjoint from $\mathcal{P}$. Below, we present an algorithm to compute a minimal (as opposed to smallest) UBS, for use in projected model counting.

4.2 Algorithm to compute projected count using UBS

We now describe an algorithm to compute a minimal UBS for a given CNF formula φ and projection set $\mathcal{P}$. We draw our motivation from Padoa's theorem [24], which provides a necessary and sufficient condition for a variable in the support of φ to be functionally determined by other variables in the support.

Let $\mathsf{Sup}(\varphi) = X = \{x_1, x_2, \dots x_t\}$; we also write $\varphi(X)$ to denote this. We create another set of *fresh* variables $X' = \{x'_1, x'_2, \dots x'_t\}$. Let $\varphi(X \mapsto X')$ represent the formula where every $x_i \in X$ in φ is replaced by $x'_i \in X'$.

Lemma 2 (Padoa's Theorem [24]). *Let $\psi(X, X', i)$ be defined as $\varphi(X) \wedge \varphi(X \mapsto X') \wedge \bigwedge_{\substack{j=1 \\ j \neq i}}^{t} (x_j \Leftrightarrow x'_j) \wedge x_i \wedge \neg x'_i$. The variable x_i is defined by $X \setminus \{x_i\}$ in the formula φ iff $\psi(X, X', i)$ is unsatisfiable.*

Padoa's theorem has been effectively used in state-of-the-art hashing-based projected model counters such as ApproxMC4 [27] to determine small independent supports of given projection sets. In our setting, we need to modify the formulation since we seek to compute an upper bound support.

Given $\mathcal{P}$, we first partition $X = \mathsf{Sup}(\varphi)$ into sets J, D and Q as follows. The set J contains variables already determined to be in a minimal UBS of $\mathcal{P}$ in φ. The set D contains variables not necessarily in a minimal UBS of $\mathcal{P}$ in φ obtainable by adding elements from Q to J. Finally, Q contains all other variables in X.

Initially, J and D are empty sets, and $Q = X$. As the process of computation of a minimal UBS proceeds, we maintain the invariant that $J \cup Q$ is a UBS (not necessarily minimal) of $\mathcal{P}$ in φ. Notice that this is trivially true initially.

Let z be a variable in Q for which we wish to determine if it can be added to the partially computed minimal UBS J. In the following discussion, we use the notation $\varphi(J, Q \setminus \{z\}, D, z)$ to denote φ with its partition of variables, and with z specially identified in the partition Q. Recalling the definition of UBS from Section 2, we observe that if z is not part of a minimal UBS containing J, and if $J \cup Q$ is indeed a UBS of $\mathcal{P} \in \varphi$, then as long as values of variables other than z in $J \cup Q$ are kept unchanged, the projection of a satisfying assignment of φ on $\mathcal{P}$ must also stay unchanged. This suggests the following check to determine if z is not part of a minimal UBS containing J.

Define $\xi(J, Q \setminus \{z\}, D, z, D', z')$ as $\varphi(J, Q \setminus \{z\}, D, z) \wedge \varphi(J, Q \setminus \{z\}, D', z') \wedge \bigvee_{x_i \,\in\, \mathcal{P} \cap (D \cup \{z\})} (x_i \not\Leftrightarrow x'_i)$, where D' and z' represent fresh and renamed instances of variables in D and z, respectively. If ξ is unsatisfiable, we know that as long as the values of variables in $J \cup (Q \setminus \{z\})$ are kept unchanged, the projection of the satisfying assignment of φ on $\mathcal{P}$ cannot change. This allows us to move z from the set Q to the set D.

Theorem 3. *If $\xi(J, Q \setminus \{z\}, D, z, D', z')$ is unsatisfiable, then $J \cup (Q \setminus \{z\})$ is a UBS of $\mathcal{P}$ in φ.*

The proof of Theorem 3 is deferred to the extended version [**?**]. The above check suggests a simple algorithm for computing a minimal UBS. We present the pseudocode of our algorithm for computing UBS below.

After initializing J, Q and D, FindUBS chooses a variable $z \in Q$ and checks if the formula ξ in Theorem 3 is unsatisfiable. If so, it adds z to D and removes it from Q. Otherwise, it adds z to J. The algorithm terminates when Q becomes empty. On termination, J gives a minimal UBS of $\mathcal{P}$ in φ. The strategy for choosing the next z from Q, implemented by sub-routine ChooseNextVar,

Algorithm 1 FindUBS($\varphi, \mathcal{P}$)

```
1: J ← ∅; Q ← Sup(φ); D ← ∅;
2: repeat
3:     z ← ChooseNextVar(Q);
4:     ξ ← ( φ(J, Q \ z, D, z) ∧ φ(J, Q \ z, D′, z′) ∧ ⋁_{x_i ∈ P∩(D∪{z})} ¬(x_i ⇔ x′_i) );
5:     if ξ is UNSAT then
6:         D ← D ∪ {z};
7:     else
8:         J ← J ∪ {z};
9:     Q ← Q \ {z};
10: until Q is ∅;
11: return J;
```

clearly affects the quality of UBS obtained from this algorithm. We require that ChooseNextVar(Q) return a variable from Q as long as $Q \neq \emptyset$. Choosing z from outside $\mathcal{P}$ gives a UBS that is the same as an IS of $\mathcal{P}$ in φ. In our experiments, we therefore bias the choice of z to favour those in $\mathcal{P}$.

In our prototype implementation, ChooseNextVar chooses variables from within $\mathcal{P}$ before variables outside $\mathcal{P}$. Note that this policy heuristically prioritizes removal of variables in $\mathcal{P}$ from the set J. To see why this is so, suppose $x_1 \leftrightarrow x_2$ is entailed by φ, and $x_1 \in \mathcal{P}$ while $x_2 \notin \mathcal{P}$. Suppose neither x_1 nor x_2 have been chosen so far. If we first choose x_1 as z, the formula ξ in line 4 of Algorithm 1 will be UNSAT, and x_1 will be moved to D and finally x_2 will be added to J (and hence to UBS). However, if we first choose x_2 as z, x_2 will be moved to D while x_1 will subsequently get added to J, and hence to UBS. We hope to leave x_2 (outside $\mathcal{P}$) in UBS and thereby first choose x_1 (within $\mathcal{P}$).

We further use an incidence-based heuristic to prioritize variables within $\mathcal{P}$, or outside $\mathcal{P}$ (after all variables in $\mathcal{P}$ have been considered). The incidence for each variable is defined as the number of clauses containing the variable or its negation in the given CNF. ChooseNextVar always returns the variable with the smallest incidence (within $\mathcal{P}$, or outside $\mathcal{P}$, as the case may be) that has not been considered so far. This is based on our observation that these variables often do not belong to upper bound support in practice.

We now state some key properties of Algorithm FindUBS. All proofs are deferred to the extended version [?].

Lemma 3. *There exists a minimal UBS $\mathcal{U}^*$ of $\mathcal{P}$ in φ such that $J \subseteq \mathcal{U}^* \subseteq J \cup Q$, where J and Q refer to the respective sets at the loop head (line 2) of Algorithm 1.*

Theorem 4. *Algorithm 1, when invoked on φ and $\mathcal{P}$, terminates and computes a minimal UBS of $\mathcal{P}$ in φ.*

The overall algorithm for computing an upper bound of the projected model count of a CNF formula using UBS is shown in Algorithm 2. This algorithm takes a timeout parameter τ_{pre} to limit the time taken for computing a UBS $\mathcal{U}$ using algorithm FindUBS. If FindUBS times out, it uses the projection set $\mathcal{P}$ itself for $\mathcal{U}$. It also invokes a PAC-style projected model counter ComputeCount to estimate the count of φ projected on $\mathcal{U}$.

Algorithm 2 $\mathsf{UBCount}(\varphi, \mathcal{P}, \varepsilon, \delta, \tau_{\mathrm{pre}})$

1: $\mathcal{U} \leftarrow$ FindUBS$(\varphi, \mathcal{P})$ with timeout τ_{pre};
2: **if** call to FindUBS times out **then**
3: $\quad \mathcal{U} \leftarrow \mathcal{P}$;
4: **return** $\mathsf{ComputeCount}(\varphi, \mathcal{U}, \varepsilon, \delta)$

Theorem 5. *Given a CNF formula* φ*, a projection set* $\mathcal{P}$*, timeout parameter* $\tau_{\mathrm{pre}} > 0$*, parameters* ε (> 0) *and* δ $(0 < \delta \leq 1)$*, and given access to a* (ε, δ)*-PAC projected counter* ComputeCount*, suppose Algorithm* UBCount *returns a count* c*. Then for every choice of sub-routine* ChooseNextVar *in Algorithm* FindUBS*, we have* $\mathsf{Pr}\left[|sol(\varphi)_{\downarrow \mathcal{P}}| \leq (1 + \varepsilon) \cdot c\right] \geq 1 - \delta$*.*

Theorem 5 provides the weakest worst-case guarantee for Algorithm UBCount, over all possible choices of sub-routine ChooseNextVar. In practice, the specifics of ChooseNextVar can be factored in to strengthen the guarantee, including PAC-style guarantees in the extreme case if ChooseNextVar always chooses variables from the projection set $\mathcal{P}$. A more detailed analysis of UBCount, taking into account the specifics of ChooseNextVar, is beyond the scope of this paper. Note, however, that despite the apparent weakness of worst-case guarantees, Algorithm UBCount consistently computes high quality bounds for projected counts in practice, as detailed in the next section.

5 Experimental Evaluation

To evaluate the practical performance of UBCount, we implemented a prototype in C++. Our prototype implementation[6] builds on Arjun [29], a state of the art independent support computation tool, which is shown to significantly improve over prior state of the art approaches for computation of independent support [21, 22]. For projected model counting, we employ the version of ApproxMC4 that was used as a winning entry to the model counting competition 2020 [14] [7]. Since all prior applications and benchmarking for approximation techniques have been presented with $\varepsilon = 0.8$ in the literature, we continue to use the same value of ε in this work. Note, however, that UBS can be used with any backend tool that computes projected model counts, and the benefits of UBS are orthogonal to those of choosing the backend projected model counter.

We use UBS+ApproxMC4 to denote the case when ApproxMC4 is invoked with the computed UBS as the projection set, while we use IS+ApproxMC4 to refer to the version of ApproxMC4 invoked with IS as the projection set.

Benchmarks. Our benchmark suite consists of 2632 instances, which are categorized into four categories: BNN, Circuit, QBF-exist and QBF-circuit. The

[6] The tool is available open-source at `https://github.com/meelgroup/arjun`.

[7] The ApproxMC4-based entry achieved 3rd place in the 2021 competition, with the tolerance for error (ε) set to 0.01. As mentioned during the competitive event presentation at the SAT 2021 conference, had ε been set to 0.05, the ApproxMC4-based entry would have indeed won the competition.

’BNN’ benchmarks are adapted from [4]. Each instance contains CNF encoding of a binarized neural network (BNN) and constraints from properties of interest such as robustness, cardinality, and parity. The projection set $\mathcal{P}$ is set to variables from a chosen layer in the BNN. The class ‘Circuit’ refers to instances from [8], which encode circuits arising from ISCAS85/ISCAS89 benchmarks conjuncted with random parity constraints imposed on output variables. The projection set, as set by authors in [8], corresponds to output variables. The ’QBF’ benchmarks are based on instances from the Prenex-2QBF track of QBFEval-17[8], QBFEval-18[9], and disjunctive decomposition [3], arithmetic [31] and factorization [3] benchmarks for Boolean functional synthesis. Each ‘QBF-exist’ benchmark is a CNF formula transformed from a QBF instance. We remove quantifiers for the (2-)QBF instances and set the projection set to the variables originally existentially quantified. The class ‘QBF-circuit’ refers to circuits synthesized using the state-of-the-art functional synthesis tool, Manthan [15]. The projection set here is set to output variables.

Our choice of benchmark categories is motivated by the observation that UBS-based approximate model counting is likely to perform well when the variables in a problem instance admit partitioning into a sequence of “layers”, with variables in each layer functionally determined by those in preceding layers. Note that this may not hold for arbitrary model counting benchmarks. We defer additional discussion on this to [**?**] for lack of space.

Experiments were conducted on a high-performance computer cluster, each node consisting of 2xE5-2690v3 CPUs with 2x12 real cores and 96GB of RAM. For each benchmark, the projected model counter with each preprocessing technique runs on a single core. We set the time limit to 5000 seconds for each of preprocessing and counting, and the memory limit to 4GB. The maximal number of conflicts in SAT solver calls during pre-processing is set to 100k. To compare runtime performance, we use PAR-2 scores, which is the de-facto standard in the SAT community. Each benchmark contributes a score that is the time in seconds taken by the corresponding tool to successfully complete execution or in case of a timeout or memory out, twice the timeout in seconds. We then calculate the average score for all benchmarks, obtaining the PAR-2 score.

We seek to answer the following research questions:

RQ 1 Does the usage of UBS enable ApproxMC4 to solve more benchmarks in comparison to the usage of IS ?

RQ 2 How does the quality of counts computed by UBS+ApproxMC4 vary in comparison to IS+ApproxMC4?

RQ 3 How does the runtime behavior of UBS+ApproxMC4 compare with that of IS+ApproxMC4?

Summary. In summary, UBS+ApproxMC4 solves 208 more instances than IS+ApproxMC4. Furthermore, while computation of UBS takes 777 more seconds, the PAR-2 score of UBS+ApproxMC4 is 817 seconds less than that of

[8] http://www.qbflib.org/qbfeval17.php
[9] http://www.qbflib.org/qbfeval18.php

IS+ApproxMC4. Finally, for all the instances where both UBS+ApproxMC4 and IS+ApproxMC4 terminated, the geometric mean of log-ratio of counts returned by IS+ApproxMC4 and UBS+ApproxMC4 is 1.32, indicating that UBS+ApproxMC4 provides good upper bound estimates. Therefore, UBS+ApproxMC4 can be used instead of IS+ApproxMC4 for applications that really care about upper bounds of projected counts.

In this context, it is worth highlighting that since there has been considerable effort in recent years in optimizing computation of IS, one would expect that further engineering efforts would lead to even more runtime savings for UBS.

Benchmarks	Total	VBS	IS+ApproxMC4	UBS+ApproxMC4
BNN	1224	868	823	823
Circuit	522	455	407	**435**
QBF-exist	607	314	156	**291**
QBF-circuit	279	152	100	**145**

Table 1. The number of solved benchmarks.

Number of Solved Benchmarks Table 1 compares the number of benchmarks solved by IS+ApproxMC4 and UBS+ApproxMC4. Observe that the usage of UBS enables ApproxMC4 to solve 435, 291, and 145 instances on Circuit, QBF-exist, and QBF-circuit benchmark sets respectively while the usage of IS+ApproxMC4 solved 407, 156 and 100 instances. In particular, UBS+ApproxMC4 solved almost twice as many instances on QBF-exist benchmarks.

The practical adoption of tools for NP-hard problems often relies on portfolio solvers. Therefore, from the perspective of practice, one is often interested in evaluating the impact of a new technique to the portfolio of existing state of the art. To this end, we often focus on Virtual Best Solver (VBS), which can be viewed as an ideal portfolio. An instance is considered to be solved by VBS if is solved by at least one solver in the portfolio. Observe that in our experiments on BNN benchmarks, while UBS+ApproxMC4 and IS+ApproxMC4 solved the same number (not same set) of instances, VBS solves 45 more instances since there were instances solved by one solver and not the other.

Time Analysis To analyze the runtime behavior, we separate the preprocessing time (computation of UBS and IS) and the time taken by ApproxMC4. Table 2 reports the mean of preprocessing time over benchmarks and the PAR-2 score for counting time. The usage of UBS reduces the PAR-2 score for counting from from 3680, 2206, 7493, and 6479 to 3607, 1766, 5238, and 4829 respectively on the four benchmark sets. Remarkably, UBS reduces PAR-2 score by over 2000 seconds on QBF-exist benchmarks and over by 1000 seconds on QBF-circuit – a significant improvement!

	Preprocessing time		PAR-2 score of counting time	
Benchmarks	IS (s)	UBS (s)	IS (s)	UBS (s)
BNN	2518	2533	3680	**3607**
Circuit	229	680	2206	**1766**
QBF-exist	70	2155	7493	**5238**
QBF-circuit	653	2541	6479	**4829**

Table 2. The mean of preprocessing time and PAR-2 score of counting time

Observe that the mean pre-processing time taken by UBS is higher than that of IS across all four benchmark classes. Such an observation may lead one to wonder whether savings due to UBS are indeed useful; in particular, one may wonder what would happen if the total time of IS+ApproxMC4 is set to 10,000 seconds so that the time remaining after IS computation can be used by ApproxMC4. We observe that even in such a case, IS+ApproxMC4 is able to solve only four more instances than Table 1. To further emphasize, UBS+ApproxMC4 where ApproxMC4 is allowed a timeout of 5000 seconds can still solve more instance than IS+ApproxMC4 where ApproxMC4 is allowed a timeout of $10,000 - t_{IS}$ where t_{IS} is time taken to compute IS with a timeout of 5000 seconds.

			IS+ApproxMC4			UBS+ApproxMC4		
Benchmarks	$\lvert X \rvert$	$\lvert \mathcal{P} \rvert$	|IS|	Time (s)	Count	|UBS|	Time (s)	Count
amba2c7n.sat	1380	1345	313	0.24+2853	$50 * 2^{65}$	73	17+1	$63 * 2^{67}$
bobtuint31neg	1634	1205	678	0.37+5000	–	417	148+16	$64 * 2^{411}$
ly2-25-bnn_32-bit-5-id-11	131	32	32	1313+3416	$94 * 2^{9}$	59	2113+1034	$63 * 2^{10}$
ly3-25-bnn_32-bit-5-id-10	131	32	32	1389+5000	–	61	2319+841	$60 * 2^{9}$
floor128	891	879	254	0.07+5000	–	256	9+6	$64 * 2^{250}$
s15850_10_10.cnf	10985	684	605	0.50+5000	–	600	41+2070	$50 * 2^{566}$
arbiter_10_5	23533	129	118	0.71+4	$64 * 2^{112}$	302	7+5000	–
cdiv_10_5	101705	128	60	102+50	$72 * 2^{50}$	–	5000+5000	–
rankfunc59_signed_64	5140	4505	1735	3+274	$43 * 2^{1727}$	–	5000+5000	–

Table 3. Performance comparison of UBS vs. IS. The runtime is reported in seconds and "–' in a column reports timeout after 5000 seconds.

Detailed Runtime Analysis Table 3 presents the results over a subset of benchmarks. Column 1 of the table gives the benchmark name, while columns 2 and 3 list the size of support X and the size of projection set $\mathcal{P}$, respectively. Columns 4-6 list the size of computed IS, runtime of IS+ApproxMC4, and model count over IS while columns 7-9 correspond to UBS. Note that the time is represented in the form $t_p + t_c$ where t_p refers to the time taken by IS (resp. UBS) and t_c refers to the time taken by ApproxMC4. We use '–' in column 6 (resp. column 9) for the cases where IS+ApproxMC4 (resp. UBS+ApproxMC4) times out.

The benchmark set was chosen to showcase different behaviors of interest: First, we observe that the smaller size of UBS for amba2c7n.sat helps UBS+ApproxMC4 while IS+ApproxMC4 times out. It is, however, worth emphasizing that the size of UBS and IS is not the only factor. To this end, observe that for the two benchmarks arising from BNN, represented in the third and fourth row, even though the size of UBS is large, the runtime of ApproxMC4 is still improved. Furthermore, in comparison to IS (which is heavily optimized in Arjun [29], our implementation for UBS did not explore engineering optimizations, which explains why UBS computation times out in the presence of the large size of support. Therefore, an important direction of future research is to further optimize the computation of UBS to fully unlock the potential of UBS.

Quality of Upper Bounds To evaluate the quality of computed upper bounds, we compare the counts computed by UBS+ApproxMC4 with those of IS+ApproxMC4 for 1376 instances where both IS+ApproxMC4 and UBS+ApproxMC4 terminated. Suppose C_{IS} and C_{UBS} denote the model count using IS and UBS respectively. The error is computed as $\mathsf{Error} = \log_2 C_{\mathrm{UBS}} - \log_2 C_{\mathrm{IS}}$, using common comparing convention for model counters. Figure 1 shows the Error distribution over our benchmarks. A point (x, y) represents $\mathsf{Error} \leq y$ on the first x benchmarks. For example, the point $(1000, 2.2)$ means that $\mathsf{Error} \leq 2.2$ on 1000 benchmarks. Overall, the geometric mean of Error is 1.32. Furthermore, for more than 67% benchmarks, Error is less than 1, and for 81% benchmarks, Error is less than 5. Only 11% benchmarks have Error larger than 10. We intend to investigate heuristics for ChooseNextVar to reduce Error in these extremal cases as part of future work. To put the significance of Error in context, we refer to the recent survey [2] comparing several partition function estimation techniques, wherein a method with Error less than 5 is considered a *reliable method*. It is known that partition function estimation reduces to model counting, and the best performing technique identified in that study relies on model counting.

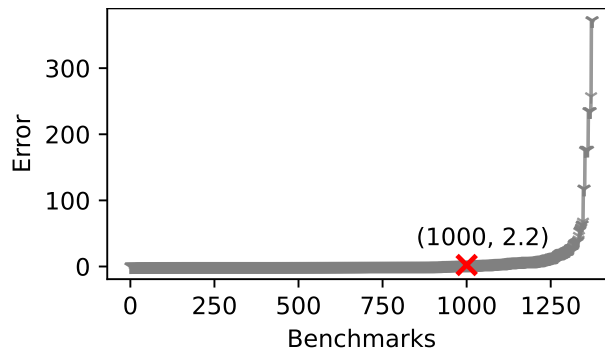

Fig. 1. Error of upper bound.

6 Conclusion

In this work, we introduced the notion of Upper Bound Support (UBS), which generalizes the well-known notion of independent support. We then observed that the usage of UBS for generation of XOR constraints allows the computation of upper bound of projected model counts. Our empirical analysis demonstrates that UBS+ApproxMC leads to significant runtime improvement in terms of the number of instances solved as well as the PAR-2 score. Since identification of the importance of IS in the context of counting led to follow-up work focused on efficient computation of IS, we hope our work will excite the community to work on efficient computation of UBS.

References

1. Achlioptas, D., Theodoropoulos, P.: Probabilistic model counting with short xors. In: Proc. of SAT. pp. 3–19 (2017)
2. Agrawal, D., Pote, Y., Meel, K.S.: Partition function estimation: A quantitative study. In: Proc. of IJCAI (8 2021)
3. Akshay, S., Chakraborty, S., John, A.K., Shah, S.: Towards parallel boolean functional synthesis. In: Proc. of TACAS. pp. 337–353 (2017)
4. Baluta, T., Shen, S., Shine, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proc. of CCS (11 2019)
5. Biondi, F., Enescu, M., Heuser, A., Legay, A., Meel, K.S., Quilbeuf, J.: Scalable approximation of quantitative information flow in programs. In: Proc. of VMCAI (1 2018)
6. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable and nearly uniform generator of sat witnesses. In: Proc. of CAV. pp. 608–622 (7 2013)
7. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Proc. of CP. pp. 200–216 (9 2013)
8. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in sat-witness generator. In: Proc. of DAC. pp. 60:1–60:6 (6 2014)
9. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In: Proc. of IJCAI (2016)
10. Chavira, M., Darwiche, A.: Compiling bayesian networks with local structure. In: IJCAI. vol. 5, pp. 1306–1312 (2005)
11. Duenas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Proc. of AAAI (2 2017)
12. Ermon, S., Gomes, C., Sabharwal, A., Selman, B.: Low-density parity constraints for hashing-based discrete integration. In: Proc. of ICML (2014)
13. Ermon, S., Gomes, C.P., Sabharwal, A., Selman, B.: Taming the curse of dimensionality: Discrete integration by hashing and optimization. In: Proc. of ICML (6 2013)

14. Fichte, J.K., Hecher, M., Hamiti, F.: The model counting competition 2020. arXiv preprint arXiv:2012.01323 (2020)
15. Golia, P., Roy, S., Meel, K.S.: Manthan: A data-driven approach for boolean function synthesis. In: Proc. of CAV (7 2020)
16. Gomes, C., Hoffmann, J., Sabharwal, A., Selman, B.: Short xors for model counting: From theory to practice. In: Proc. of SAT (2007)
17. Gomes, C., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. pp. 2293–2299 (01 2007)
18. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting: A new strategy for obtaining good bounds. In: Proc. of AAAI (2006)
19. Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. Constraints **21**(1) (9 2016)
20. Kroc, L., Sabharwal, A., Selman, B.: Leveraging belief propagation, backtrack search, and statistics for model counting. In: Proc. of CPAIOR (2008)
21. Lagniez, J.M., Lonca, E., Marquis, P.: Improving model counting by leveraging definability. In: IJCAI. pp. 751–757 (2016)
22. Lagniez, J.M., Lonca, E., Marquis, P.: Definability for model counting. Artificial Intelligence **281**, 103229 (2020)
23. Meel, K.S., Akshay, S.: Sparse hashing for scalable approximate model counting: Theory and practice. In: Proc. of LICS (7 2020)
24. Padoa, A.: Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une théorie déductive quelconque. Bibliothèque du Congrès International de Philosophie **3**, 309 (1901)
25. Sang, T., Bearne, P., Kautz, H.: Performing bayesian inference by weighted model counting. In: Proc. of AAAI. AAAI'05, vol. 1, p. 475–481 (2005)
26. Sharma, S., Roy, S., Soos, M., Meel, K.S.: Ganak: A scalable probabilistic exact model counter. In: IJCAI. vol. 19, pp. 1169–1176 (2019)
27. Soos, M., Gocht, S., Meel, K.S.: Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling. In: Proc. of CAV (7 2020)
28. Soos, M., Meel, K.S.: Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In: Proc. of AAAI (1 2019)
29. Soos, M., Meel, K.S.: Arjun: An efficient independent support computation technique and its applications to counting and sampling. arXiv preprint arXiv:2110.09026 (2021)
30. Stockmeyer, L.: The complexity of approximate counting. In: Proc. of STOC (1983)
31. Tabajara, L.M., Vardi, M.Y.: Factored boolean functional synthesis. In: Proc. of FMCAD. p. 124–131. FMCAD '17 (2017)
32. Teuber, S., Weigl, A.: Quantifying software reliability via model-counting. In: Proc. of QEST. pp. 59–79 (2021)
33. Valiant, L.G.: The complexity of computing the permanent. Theoretical Computer Science **8**(2), 189–201 (1979)
34. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM Journal on Computing **8**(3), 410–421 (1979)